PSEUDO-INSTRUÇÕES PROLOG E SEU AMBIENTE DE EXECUÇÃO PARTE I

Luiz Fernando P. de Souza
Valério Machado Dallolio

NCE-12/88

Grupo de Inteligência Artificial/UFRJ

Outubro/88

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletrônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL

Este trabalho foi parcialmente financiado com recursos do Projeto ESTRA da SID Informática S.A.

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

Índice

Parte I

Parte II

0. Sinopse.

Este relatório sintetiza parte do trabalho realizado pelo Grupo de Inteligência Artificial da Universidade Federal do Rio de Janeiro na pesquisa de implementações de compiladores Prolog. É sugerida uma implementação baseada no segundo trabalho de Warren, sendo revistos em minúcia os aspectos apresentados naquele trabalho. São resolvidos vários detalhes não mencionados por Warren e são apresentadas algumas extensões.

Além disto, abordamos cuidadosamente os mecanismos de execução da linguagem Prolog (Parte I) e expomos precisamente como utilizar as pseudo-instruções do código intermediário (Parte II).

0. Abstract.

This report synthetizes part of the research into Prolog compiler implementations that has been done by the Artificial Inteligence Group (Grupo de Inteligência Artificial) from Universidade Federal do Rio de Janeiro. In this work, we suggest an implementation based on Warren's second report. The contents of Warren's work are reviewed in detail. Several points that have not been mentioned in Warren's work are solved. And some improvements are presented.

The Prolog Language execution mechanisms are described in detail (Part I). This work also establishes precisely how to use the pseudo-code (Part II).

1. Introdução.

A linguagem Prolog foi inicialmente especificada por Alain Colmerauer do Grupo de Inteligência Artificial de Marselha por volta de 1975. Em 1981, depois do comitê japonês para computadores de quinta geração ter escolhido uma arquitetura baseada em Prolog para o desenvolvimento de seus computadores, esta linguagem tomou um novo impulso. Atualmente ela é largamente utilizada nos projetos de inteligência artificial.

O Grupo de Inteligência Artificial da UFRJ encontra-se atualmente desenvolvendo vários projetos interrelacionados. Estes projetos englobam: sistemas especialistas, linguagem natural e ambientes de desenvolvimento em I.A.. Um destes ambientes consiste de um interpretador Prolog. Os principais motivos que levaram ao desenvolvimento do interpretador são: a necessidade de um ambiente Prolog para desenvolvimento de outros projetos escritos nesta linguagem; o interesse de investigar e difundir a tecnologia de implementação do Prolog; e a existência de um grupo interessado na pesquisa de arquiteturas dedicadas.

Depois de decidida a implementação de um ambiente Prolog estabelecemos alguns requisitos que o sistema deveria apresentar. A principal característica é o uso de uma versão padrão da linguagem. Por isto especificamos para a nossa implementação uma sintaxe que englobasse a usada em Edimburgo, como descrito por Clocksin e Mellish [CLOC81]. Também incluimos em nosso sistema a maioria das facilidades encontradas no interpretador Arity [ARIT84].

A segunda característica que impomos ao nosso projeto é que o interpretador deve executar sobre a maioria dos sistemas nacionais. Para isto resolvemos que o interpretador será implementado na linguagem C, de forma independente de sistemas operacionais e arquiteturas. O interpretador deverá executar no mínimo em dois tipos de ambientes: PC-DOS e UNIX. Acreditamos que o nosso interpretador poderá ser facilmente transportado para os mais diferentes ambientes que suportem a linguagem C.

Com base neste trabalho foram desenvolvidos dois módulos principais: geração e análise. O módulo de geração inclui um módulo provisório de análise (gerado automaticamente) e é responsável pela compilação das cláusulas Prolog no código intermediário. O código gerado é então passado ao interpretador que simula a semântica de cada pseudo-instrução. Atualmente a equipe está trabalhando no módulo de análise definitivo que deverá estar pronto até o final do ano.

Após o sistema estar consolidado, teremos duas frentes de desenvolvimento a atacar. Uma consiste nas extensões da linguagem conhecidas como Prolog II. E outra consiste em adicionar ao Prolog I recursos que atualmente são desejáveis em qualquer linguagem. Alguns destes recursos são: ferramentas de interface

com o usuário, consistindo de telas, janelas e menus no vídeo além do controle do teclado; outra extensão muito requerida pelos usuários de Prolog é uma interface com algum banco de dados, notadamente uma interface padrão SQL.

O trabalho está organizado da seguinte forma. O capítulo 2 apresenta rapidamente os aspectos da linguagem tornando clara as diferenças entre Prolog e as linguagens convencionais de programação. O capítlo 3 apresenta abstratamente as estruturas de dados necessárias para suportar a execução do Prolog (Parte I). Finalmente, na Parte II do relatório, abordamos as estruturas e a representação interna utilizadas em nossa implementação.

2. Aspectos da Execução Prolog.

Este capítulo destina-se a discutir os conceitos básicos de organização e fluxo de execução do Prolog, relevantes ao funcionamento interno da máquina Prolog.

2.1. Mecanismos de Controle.

Em Prolog, como em linguagens de programação do tipo Pascal, informações devem ser guardadas em registros de ativação ao longo da execução para posterior utilização. Estes registros de ativação são denominados molduras. Mostramos neste item que tipo de informações devem ser guardadas nas molduras a fim de possibilitar o processo de resolução. No próximo capítulo discutiremos mais a fundo a organização destas informações e sua manipulação durante a execução.

2.1.1. Retorno com Sucesso.

A necessidade de guardarmos detalhes de como evoluímos do início do processamento até o estado atual advém, em parte, do fato de termos de lidar com a operação de retorno de subrotina. No caso específico do Prolog, este retorno segue-se a cada fim de procedimento com sucesso, quando então temos que transferir o controle para o procedimento que causou a chamada, o objetivo-pai, e continuarmos executando os objetivos que o compõem. Vale a pena ressaltar que devido ao retrocesso, o término de um procedimento não implica necessariamente em jogarmos fora sua moldura.

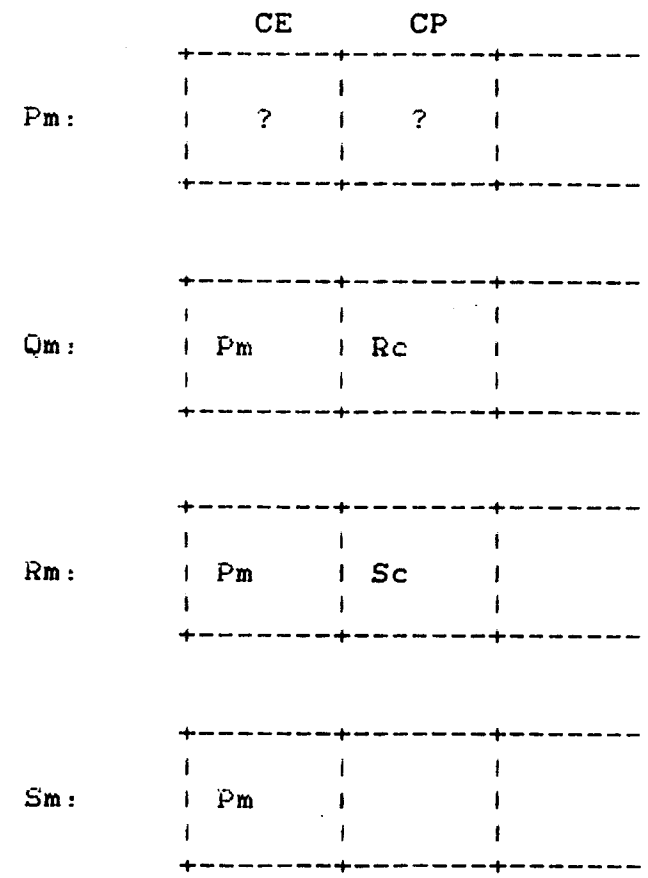
Passemos a analisar o mecanismo encarregado do retorno com sucesso, sem nos preocuparmos com parâmetros e variáveis. Utilizaremos nos próximos exemplos a seguinte convenção: "Xc" é um rótulo para o código da cláusula "X" e "Xm" um rótulo para sua moldura. Estudemos então o seguinte pedaço de programa:

```
Pc:     p :- q, r, s.

Qc:     q :- ...

Rc:     r :- ...

Sc:     s :- ...
```

Sem muito rigor podemos traduzi-lo para:

```
Pc:     p :- call( q ), call( r ), call( s ).

Qc:     q :- ...

Rc:     r :- ...

Sc:     s :- ...
```

Suponha agora que estamos na cláusula "q" e, tendo esta sido executada com sucesso, precisamos executar o próximo objetivo do corpo da cláusula "p", que causou sua ativação. Concluímos que precisamos ter na moldura dos procedimentos um ponteiro para o objetivo de continuação (CP). Para o procedimento "s" não há objetivos de continuação, ao seu término devemos restaurar o contexto do procedimento "p", isto é, devemos tornar ativa sua moldura. Cada moldura, contém além de CP, um ponteiro para a moldura do objetivo-pai, denominado CE.



2.1.2. Retrocesso.

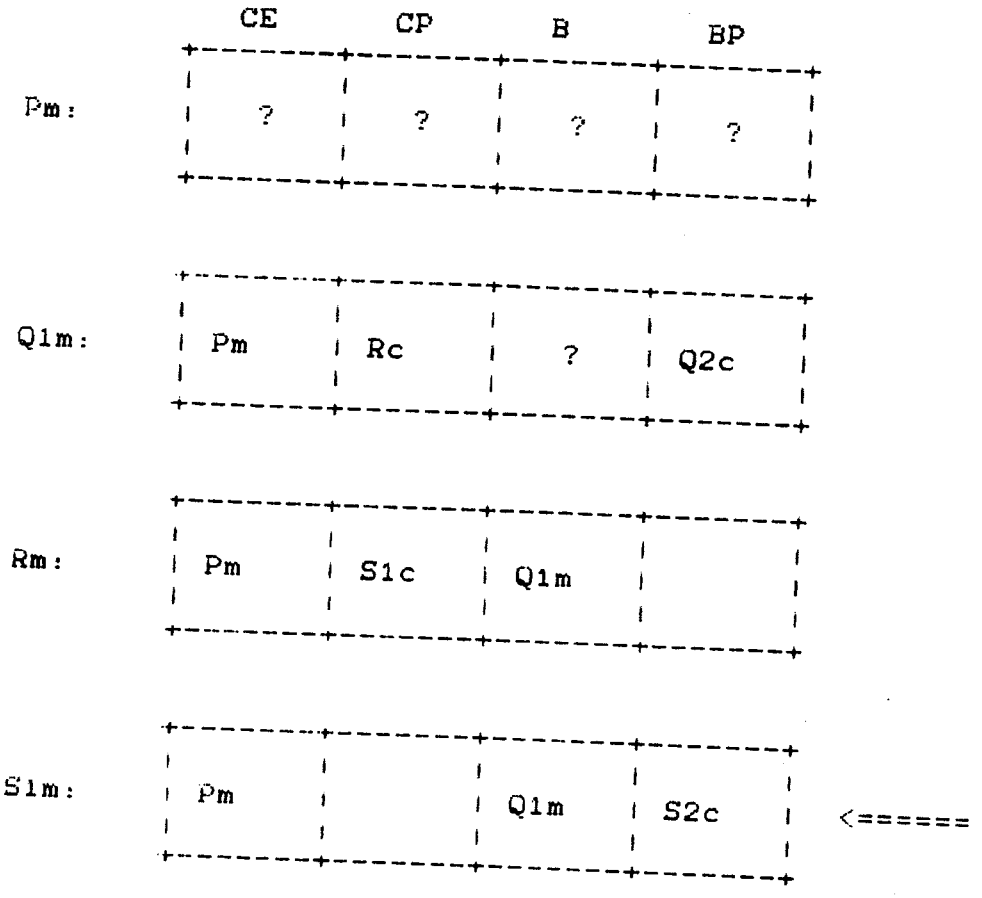
Quando uma falha ocorre, devemos retroceder, isto é, retornar ao passo mais recentemente executado para o qual haja alternativas não testadas retomando, a partir deste ponto, a execução. Neste caso, duas situações podem ocorrer:

1) Se há alternativa dentro do próprio procedimento devemos então ativá-la, necessitando para isso de um ponteiro (BP) para o código do próximo ponto de entrada no procedimento em questão.

2) Caso o procedimento corrente não possua alternativas, retrocedemos ao procedimento mais próximo da cláusula rejeitada que ainda possua alternativas. Para isso precisamos de um ponteiro (B) que nos informe a moldura do procedimento mais recentemente executado, que possua alternativas, para que através deste possamos acessar o BP correspondente.

Estendendo o último exemplo, procedimentos "q" e "s" com alternativas, e supondo primeiramente uma execução normal (sem retrocesso) temos:

```
Pc:     p :- q, r, s.

Q1c:    q :- ...

Q2c:    q :- ...

Rc:     r :- ...

S1c:    s :- ...

S2c:    s :- ...
```



Se neste ponto ocorresse uma falha na cláusula "s", estaríamos na situação um (1) de retrocesso e teríamos apenas que alterar o campo BP da moldura (assinalado acima):

	CE	CP	B	BP
Pm:	?	?	?	?

Q1m:	Pm	Rc	?	Q2c

Rm:	Pm	S1c	Q1m	

S2m:	Pm		Q1m	

Um novo retrocesso em "s", situação dois (2) de retrocesso, desalocaria as molduras de "r" e "s", nos levando a seguinte configuração:

	CE	CP	B	BP
Pm:	?	?	?	?

Q2m:	Pm	Rc	?	

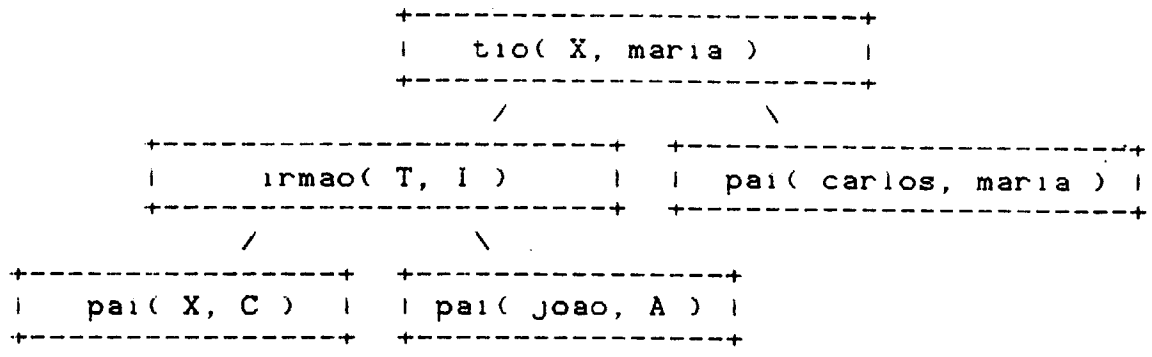
3. Ambiente de Execução.

Neste capítulo, completaremos o conjunto das informações armazenadas em molduras, mostrando como estas estarão organizadas em nossa implementação.

3.1. Organização das Molduras.

Como foi dito, uma cláusula do Prolog é basicamente uma seqüência de chamadas a procedimentos. Assim sendo, a execução de um programa Prolog pode ser representada através de uma estrutura em árvore, como o exemplo abaixo:

```
pai( joao, jose ).
pai( joao, carlos ).
pai( jose, augusto ).
pai( carlos, maria ).

tio( T, S ) :- irmao( T, I ), pai( I, S ).

irmao( C, A ) :- pai( X, C ), pai( X, A ).

?- tio( X, maria ).
X = jose
```



Cada nó da árvore deve ser visto como a moldura do objetivo correspondente, e os filhos deste nó como as molduras dos objetivos subordinados. Em linguagens convencionais estas molduras são desalocadas à medida que os procedimentos terminam. Entretanto, no Prolog, as molduras dos procedimentos já concluídos devem continuar existindo para permitir um eventual retrocesso.

Podemos observar que as molduras são criadas como se percorrêssemos a árvore em pré-ordem. O retrocesso, por sua vez, procura por cláusulas com alternativas a partir das molduras mais recentemente criadas para as mais antigas, ou seja, na ordem inversa em que foram criadas. Não é difícil perceber que este mecanismo se adapta muito bem à estrutura de pilha, na qual a moldura mais ao topo será sempre aquela sobre a qual o retrocesso deve agir.

Inicialmente podemos considerar que cada moldura contém, além das informações normalmente encontradas em registros de ativação tais como variáveis, parâmetros, ponto de retorno e elos dinâmicos e estáticos, aquelas necessárias ao funcionamento do Prolog.

3.2. Mecanismos de Desunificação : Trilha.

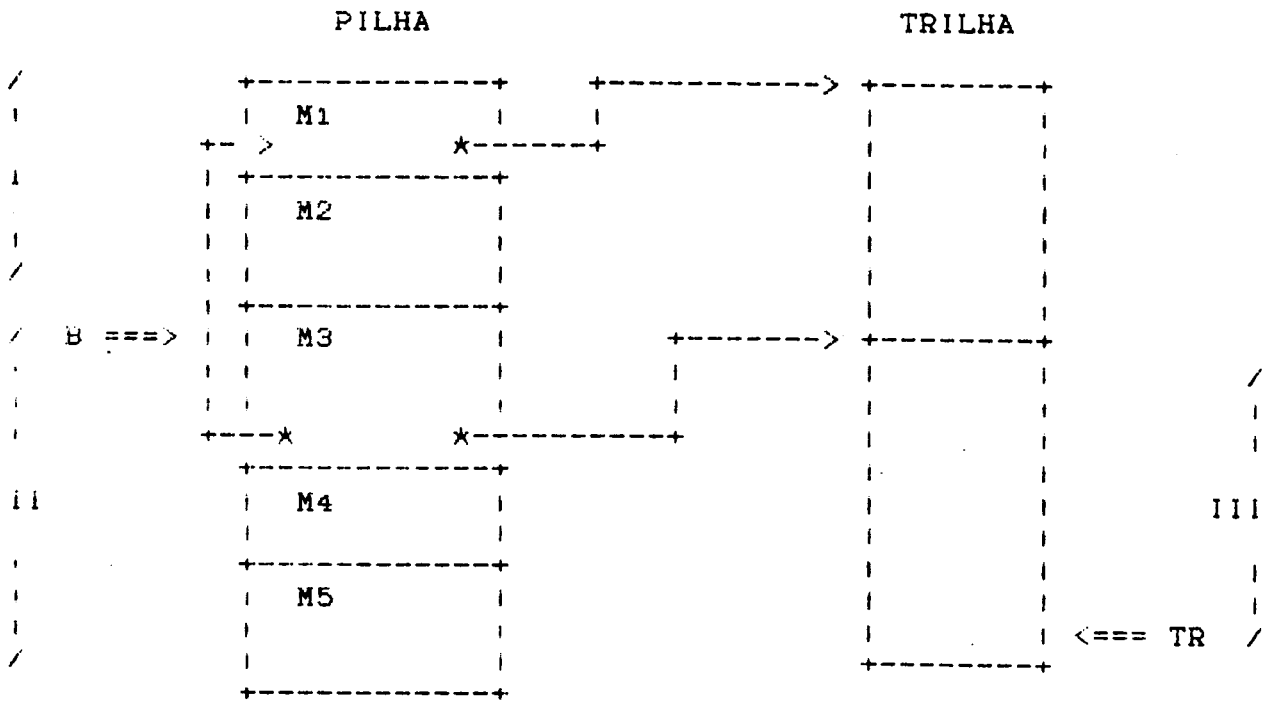
Uma vez que o retrocesso localizou o procedimento mais recentemente ativado para o qual existe alternativa, a execução deve prosseguir a partir desta, considerando a cláusula alternativa. Para que isto seja possível, as variáveis devem retornar ao que eram no momento da chamada anterior. Para que possamos recuperar os parâmetros, todo procedimento que possuir mais de uma cláusula deve copiar em sua moldura os parâmetros que lhe foram passados. No caso de procedimentos com apenas uma cláusula, isto não é necessário já que, não possuindo alternativas, este não será considerado pelo retrocesso.

No caso de variáveis devemos nos preocupar com aquelas que foram alteradas após a ativação do procedimento em questão. Vale lembrar que só as variáveis livres podem ser alteradas. Assim sendo, necessitamos de uma estrutura de dados, a Trilha, que nos permita saber quais variáveis deixaram de ser livres desde a ativação de determinado procedimento. Quando ocorre um retrocesso, estas variáveis devem voltar a ser livres, num processo a que chamamos desunificação. Na Trilha são empilhadas referências para as variáveis que são alteradas. Em cada moldura de procedimento colocamos um ponteiro para o topo da Trilha, no momento da ativação. Assim, as variáveis alteradas desde a ativação de um determinado procedimento são aquelas cujas referências estão entre o topo atual da Trilha, TR, e o ponteiro armazenado na moldura do procedimento.

Nem todas as variáveis alteradas precisam ter suas referências colocadas na Trilha. Somente aquelas que estejam em molduras mais antigas que a moldura do procedimento sobre o qual o retrocesso atuará, visto que as demais molduras deixarão de existir quando houver o retrocesso. Para isto, mantemos um registro, B, apontando para a moldura do procedimento mais recente que possua alternativas. Obviamente, sempre que um novo procedimento, que possua alternativas, é ativado, salvamos B na sua moldura e o atualizamos, formando desta forma uma cadeia de molduras a serem

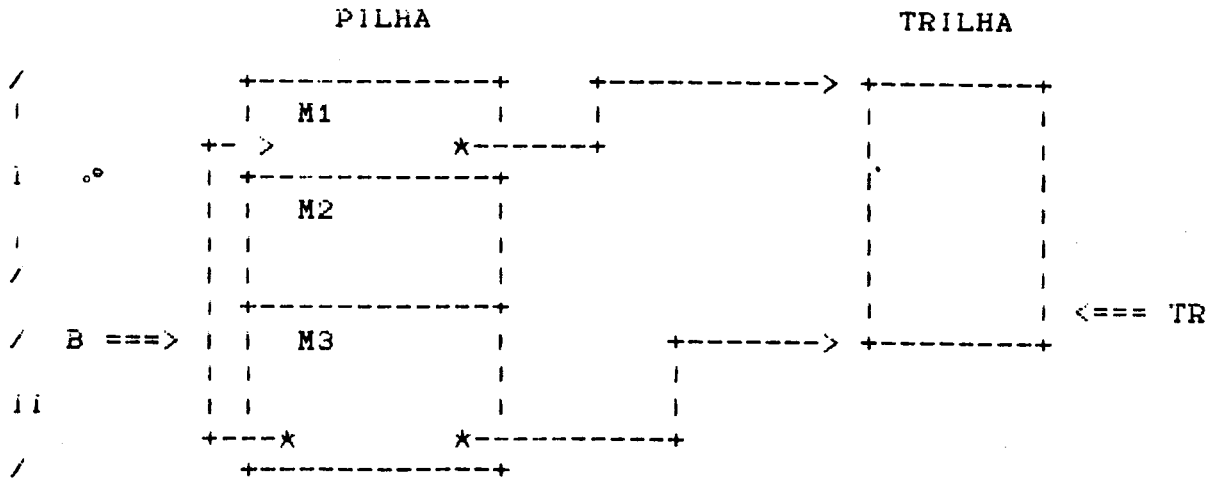
consideradas pelo retrocesso.

Assim sendo, na figura abaixo, variáveis na área I devem ter suas referências colocadas na Trilha, caso sejam alteradas. Para variáveis na região II isto não é necessário.

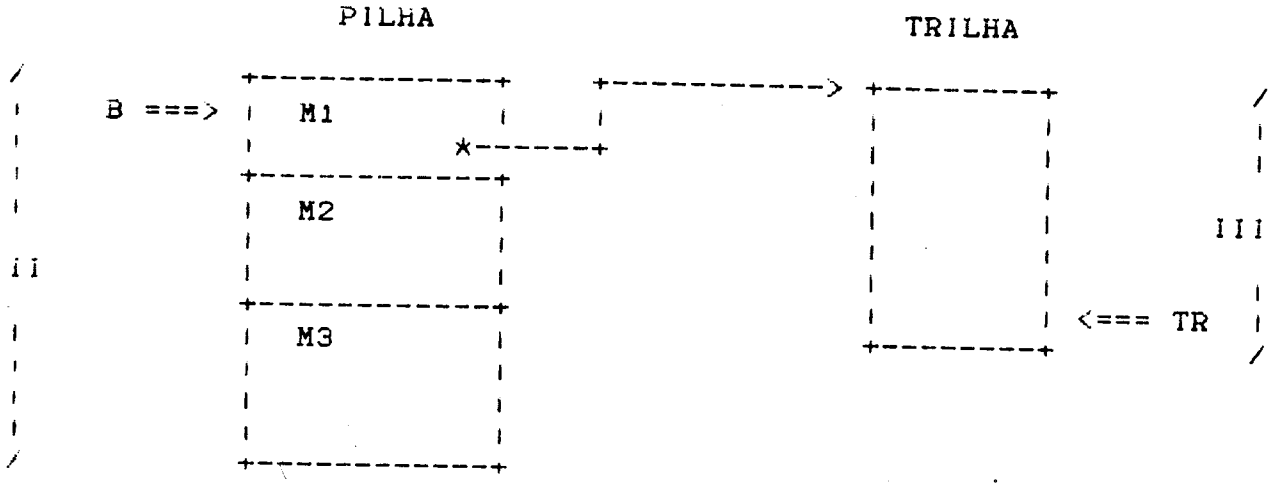


No caso de retrocesso todas as variáveis com referência na área III devem ser desunificadas. Ficamos, neste caso, com uma das seguintes configurações:

1) No caso de M3 ainda possuir alternativas além da que vamos considerar.



2) No caso de M3 não possuir alternativas além da que vamos considerar.



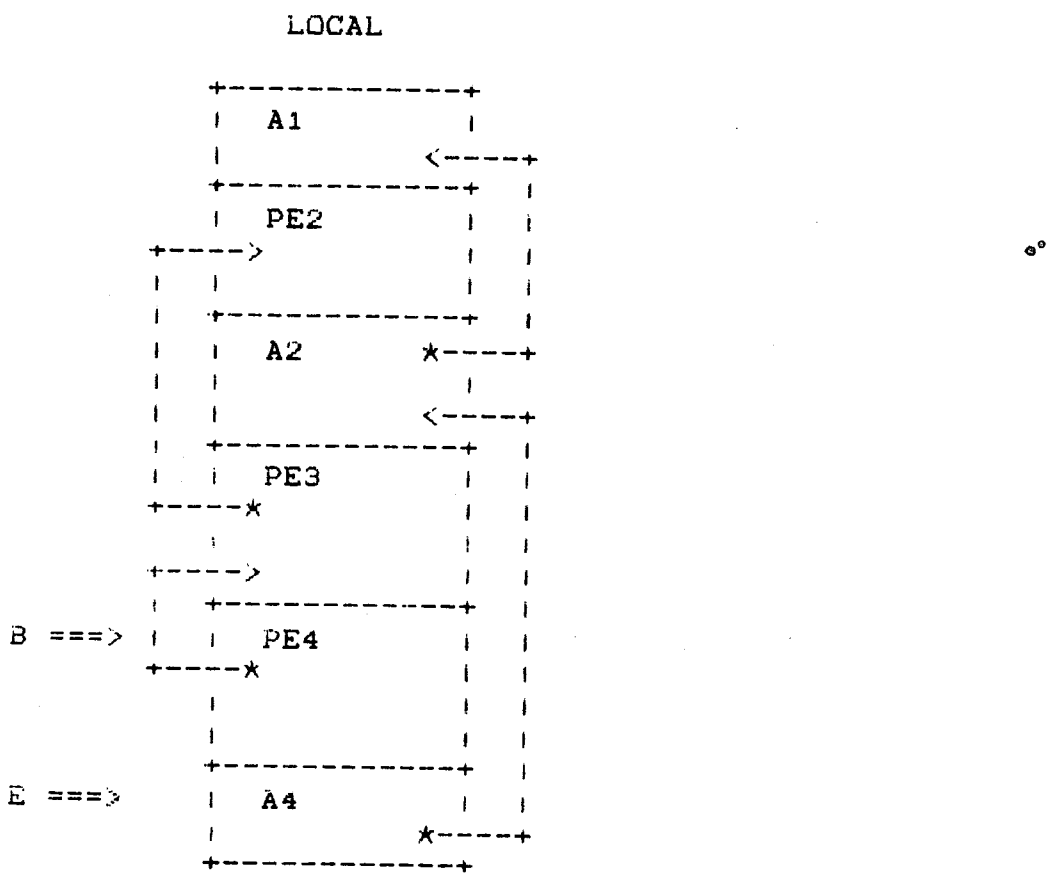
3.3. Pilha Local.

Como acabamos de ver, procedimentos compostos de mais de uma cláusula devem guardar em sua moldura as seguintes informações adicionais: copia dos parâmetros, ponteiro para o topo da trilha, o conteúdo anterior ao registro B e o ponteiro para a próxima alternativa. Além destes, devemos copiar alguns registros que formam o contexto do procedimento, para que possamos restaurá-los em caso de retrocesso. Dentre eles CP e E, como vimos no capítulo anterior. Para procedimentos de uma só cláusula algumas dessas informações não são necessarias. Logo, as molduras são particionadas em dois tipos: pontos de escolha e ambientes.

Os pontos de escolha possuem informações necessárias ao retrocesso e são criados para procedimentos que possuem mais de uma cláusula (alternativas). Para que, em caso de retrocesso, consigamos refazer a chamada original, os parâmetros passados nesta chamada devem ser guardados no ponto de escolha do procedimento. Os pontos de escolha só são descartados em caso de retrocesso que esgote as alternativas ou através do pré-definido "corte", como veremos mais tarde.

Os ambientes contém informações normalmente encontradas em registros de ativação e são criados somente para procedimentos que possuam mais de um objetivo. Vale observar que, procedimentos Prolog, que não possuem alternativas, são bastante semelhantes aos procedimentos de linguagens de programação convencionais e, podem ser descartados ao término do procedimento. Somente em dois casos especiais os pontos de escolha são desalocados. Segue-se que somente os ambientes mais ao topo na pilha Local que o último ponto de escolha, podem ser descartados ao final de sua cláusula.

Atualmente temos o seguinte possível aspecto para a Pilha Local:



Os pontos aqui descritos ficarão mais claros no exemplo a seguir.

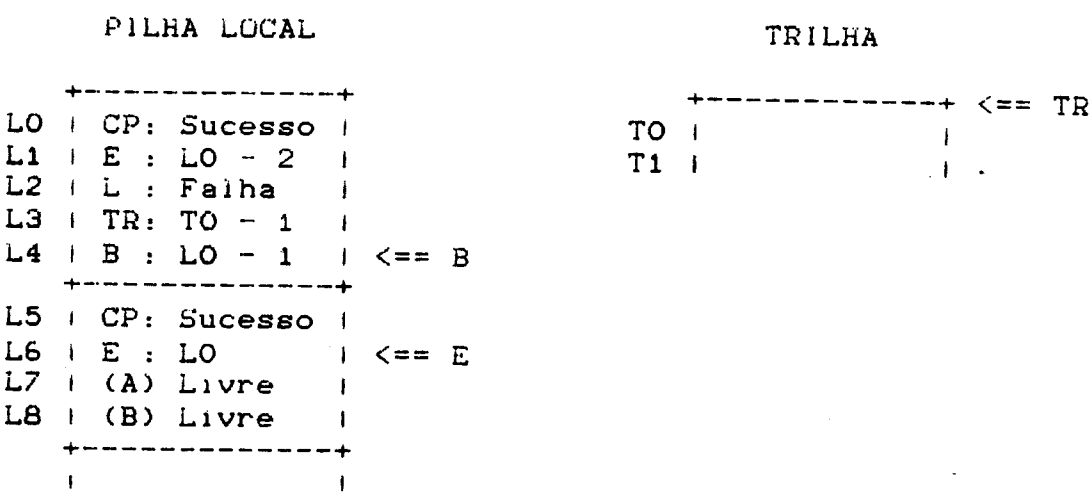
3.3.1. Exemplo de Funcionamento da Pilha Local e da Trilha.

Exemplo:

```
1- portavel( P, varios ) :- escrito( P, c ).

2- portavel( P, msdos ) :- escrito( P, pascal ),
                           sistema( P, msdos ).
3- portavel( P, unix ) :- escrito( P, pascal ),
                          sistema( P, unix ).

4- escrito( interpretador, c ).

5- escrito( tedmos, pascal ).

6- pronto( tedmos ).

7- sistema( tedmos, msdos ).

8- disponivel( X, S ) :- pronto( X ),
                         portavel( X, S ).

9- ?- disponivel( A, B ).
```

(1) Configuração inicial.

Ativação de "disponivel(A, B)".



(2) Entrada em "disponivel(X, S)".

Unificação da pergunta com a cabeça da cláusula. Variáveis mais recentes, "X" e "S", referenciam variáveis mais antigas, "A" e "B".

PILHA LOCAL

```
    +--------------+
L0  | CP: Sucesso  |
L1  | E : L0 - 2   |
L2  | L : Falha    |
L3  | TR: T0 - 1   |
L4  | B : L0 - 1   | <== B
    +--------------+
L5  | CP: Sucesso  |
L6  | E : L0 - 2   |
L7  | (A) Livre    |
L8  | (B) Livre    |
    +--------------+
L9  | CP: Sucesso  |
L10 | E : L6       | <== E
L11 | (X) L7       |
L12 | (S) L8       |
    +--------------+
    |              |
```

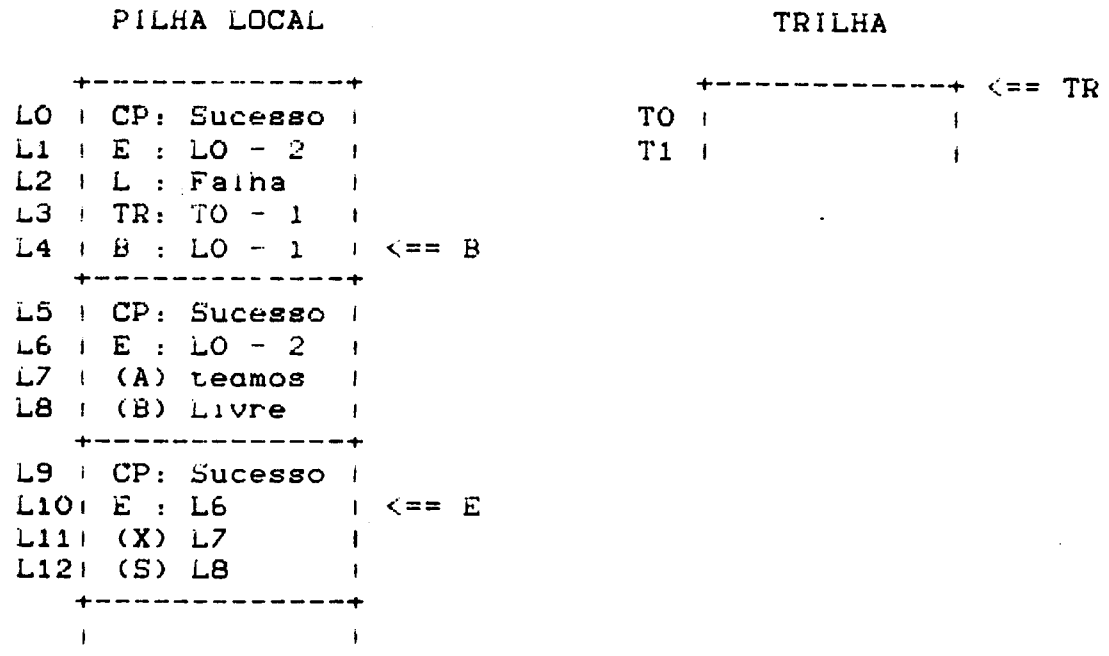
TRILHA

```
    +--------------+ <== TR
T0  |              |
T1  |              |
```

(3) Ativação de "pronto(X)".

A unificação da variável "X", "A" na verdade, com a constante "teamos" não faz com que a variável tenha sua referência colocada na Trilha pois "A" está mais ao topo que o ponto de escolha mais recente.

O procedimento "pronto" não cria ambiente, pois não possui mais de um objetivo, e também não cria ponto de escolha já que é composto por apenas uma cláusula.



(4) Ativação de "portavel(tedmos, S)".

É criado um ponto de escolha onde são copiados os parâmetros, e a alternativa é a segunda cláusula de portável. Como vimos, o ambiente de disponível pode ser descartado, pois não havia ponto de escolha mais ao topo que este.

A unificação com a cabeça da cláusula faz com que "S", na verdade "B", passe a valer "varios". Uma vez que esta variável está mais abaixo na pilha que o ponto de escolha mais recente, colocamos sua referência na Trilha.

PILHA LOCAL

```
    +--------------+
L0  | CP: Sucesso  |
L1  | E : L0 - 2   |
L2  | L : Falha    |
L3  | TR: T0 - 1   |
L4  | B : L0 - 1   |
    +--------------+
L5  | CP: Sucesso  |
L6  | E : L0 - 2   | <== E
L7  | (A) tedmos   |
L8  | (B) varios   |
    +--------------+
L9  | (X) L7       |
L10 | (S) L8       |
L11 | CP: Sucesso  |
L12 | E : L6       |
L13 | L : claus. 2 |
L14 | TR: T0 - 1   |
L15 | B : L4       | <== B
    +--------------+
    |              |
```

TRILHA

```
    +--------------+
T0  | L8           | <== TR
T1  |              |
```

(5) Ativação de "escrito(tedmos, c)".

É criado um ponto de escolha visto que o procedimento tem duas cláusulas. Como a ativação não unifica com a cabeça da primeira cláusula do procedimento "escrito", usamos este ponto de escolha logo em seguida, desviando para o endereço indicado em L. Como o valor de TR guardado no ponto de escolha coincide com o valor atual do registrador TR, concluimos que nenhuma variável deve ser desunificada.

PILHA LOCAL

L0	CP: Sucesso	
L1	E : L0 - 2	
L2	L : Falha	
L3	TR: T0 - 1	
L4	B : L0 - 1	
L5	CP: Sucesso	
L6	E : L0 - 2	<== E
L7	(A) tedmos	
L8	(B) varios	
L9	(X) L7	
L10	(S) L8	
L11	CP: Sucesso	
L12	E : L6	
L13	L : claus. 2	
L14	TR: T0 - 1	
L15	B : L4	
L16	tedmos	
L17	c	
L18	CP: Sucesso	
L19	E : L6	
L20	L : claus. 5	
L21	TR: T0	
L22	B : L15	<== B

TRILHA

T0	L8	<== TR
T1		

(6) Ativação da segunda cláusula de "escrito".

Utilizamos para isso os parâmetros salvos no ponto de escolha do procedimento. Como não há mais alternativas a considerar, o ponto de escolha é removido.

A cláusula ativada também não unifica com a chamada, ocorrendo um novo retrocesso, utilizando desta vez as informações contidas no ponto de escolha criado por "portavel".

PILHA LOCAL

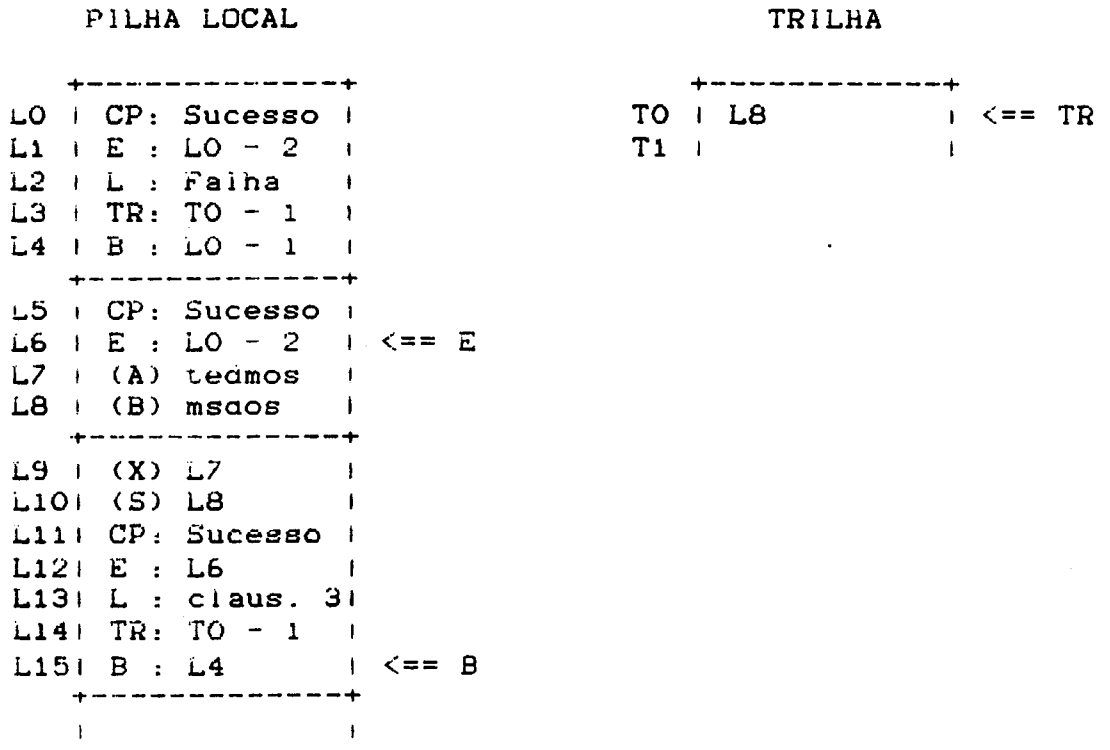
```
    +--------------+
L0  | CP: Sucesso  |
L1  | E : L0 - 2   |
L2  | L : Falha    |
L3  | TR: T0 - 1   |
L4  | B : L0 - 1   |
    +--------------+
L5  | CP: Sucesso  |
L6  | E : L0 - 2   | <== E
L7  | (A) teamos   |
L8  | (B) varios   |
    +--------------+
L9  | (X) L7       |
L10 | (S) L8       |
L11 | CP: Sucesso  |
L12 | E : L6       |
L13 | L : claus. 2 |
L14 | TR: T0 - 1   |
L15 | B : L4       | <== B
    +--------------+
    |              |
```

TRILHA

```
    +-------------+
T0  | L8          | <== TR
T1  |             |
```

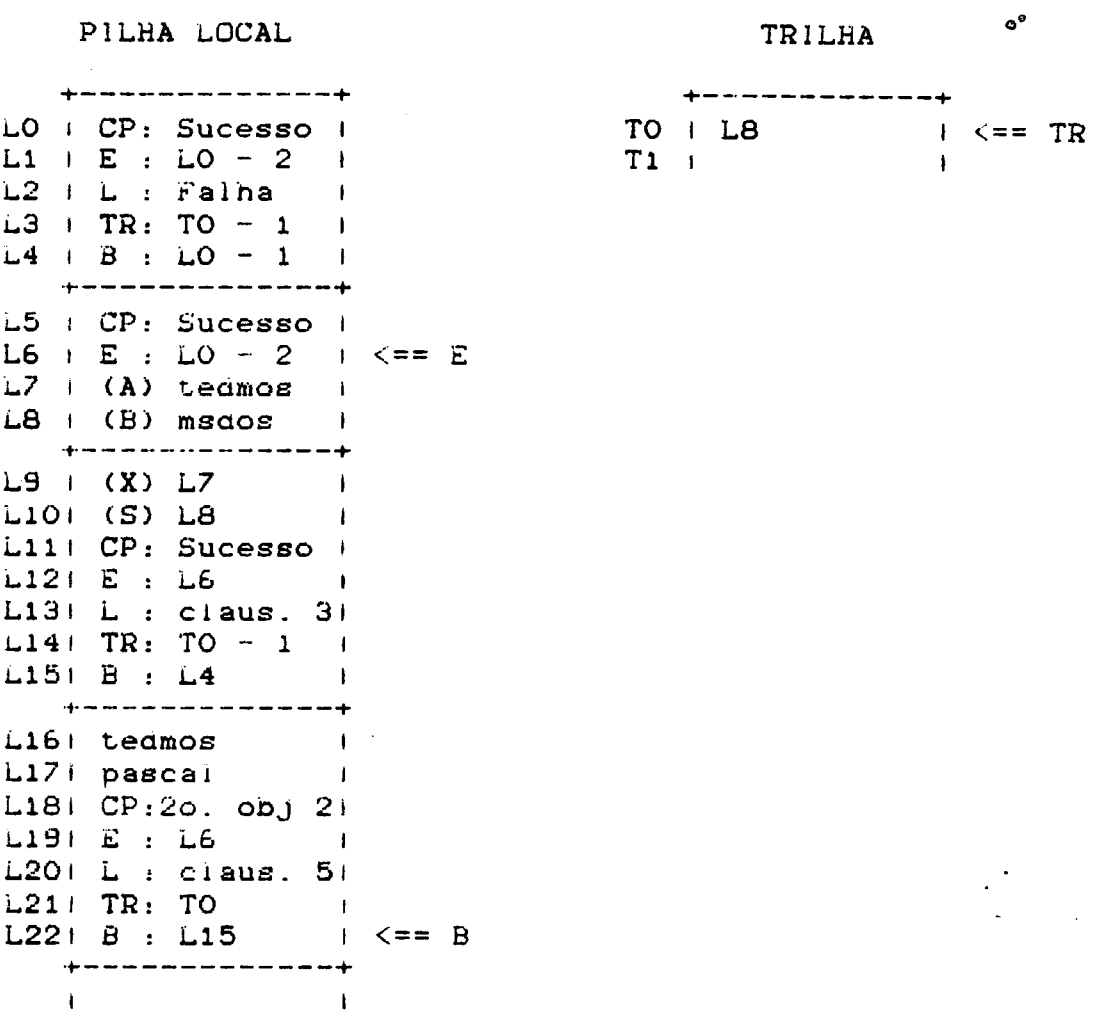
(7) Ativação da segunda cláusula de "portavel".

O retrocesso em cima do ponto de escolha criado por "portavel" faz com que a variável cuja referência está na Trilha saja desunificada, tornada livre. A unificação da chamada com a cabeça da segunda cláusula do predicado atribui a variável "S", "B" na verdade, a constante "msaos". Uma nova referência para "B" é colocada na Trilha. É feito então, a chamada ao procedimento "escrito".



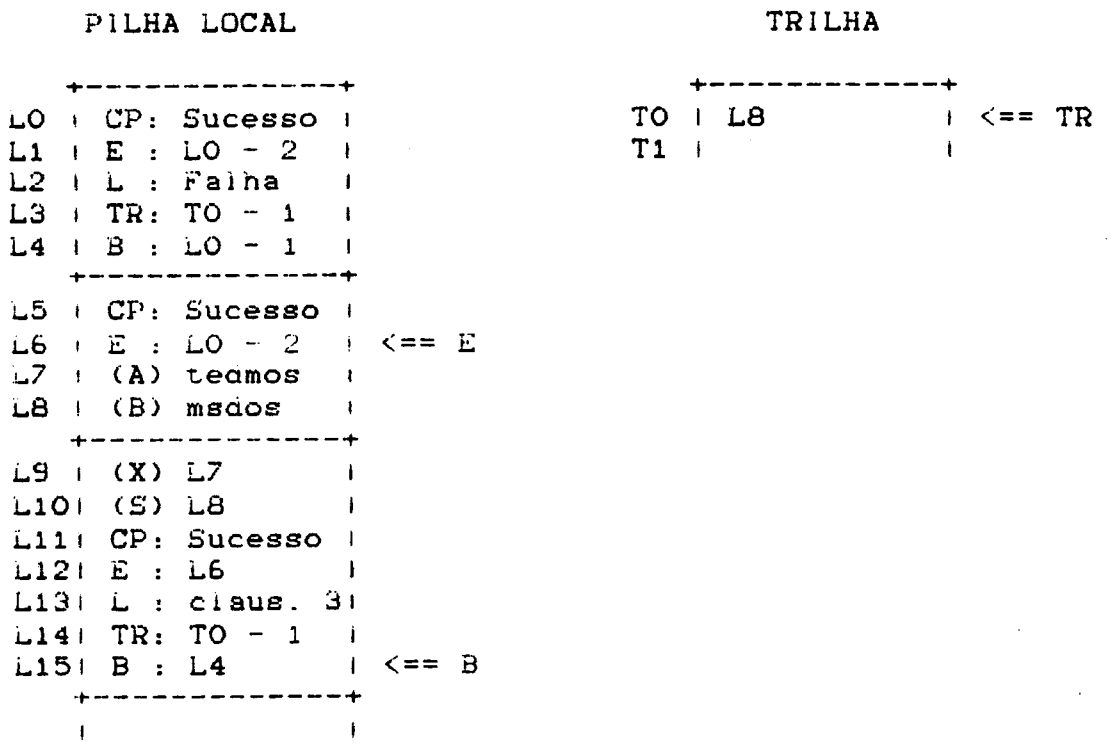
(8) Entrada em "escrito(tedmos, pascal)".

A primeira cláusula deste objetivo não unifica com a chamada. Continuamos então pela cláusula alternativa, que unifica com a chamada, satisfazendo este objetivo. O campo CP do ponto de escolha nos dá o objetivo de continuação. O ponto de escolha de "escrito" deve ser desalocado pois não há mais alternativas a considerar.



(9) Ativação de "sistema(teamos, msdos)".

A chamada unifica com a cabeça da cláusula. Este procedimento não cria nem ponto de escolha nem ambiente e a execução prossegue então pelo ponteiro de continuação, CP, do ambiente, que no caso contém sucesso. Logo, a execução termina reportando o valor das variáveis "A" = "teamos" e "B" = "msdos".



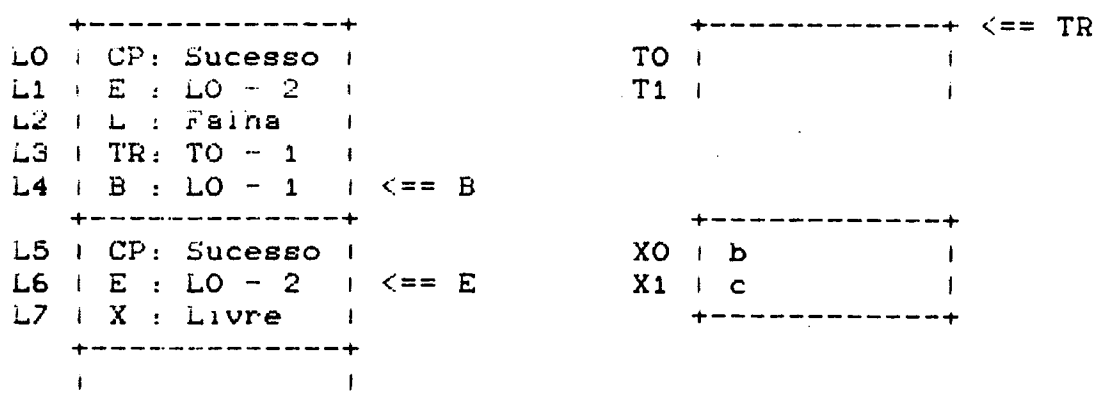
3.4. Pilha Global.

Como veremos no exemplo abaixo, apenas a pilha Local e a Trilha não são suficientes para suportarmos listas e estruturas. Nestes casos veremos que surgem referências pendentes devido a desalocação dos ambientes. Para efeito de simplificação no exemplo, não nos preocuparemos com o armazenamento das estruturas, mas sim com as referências para as variáveis.

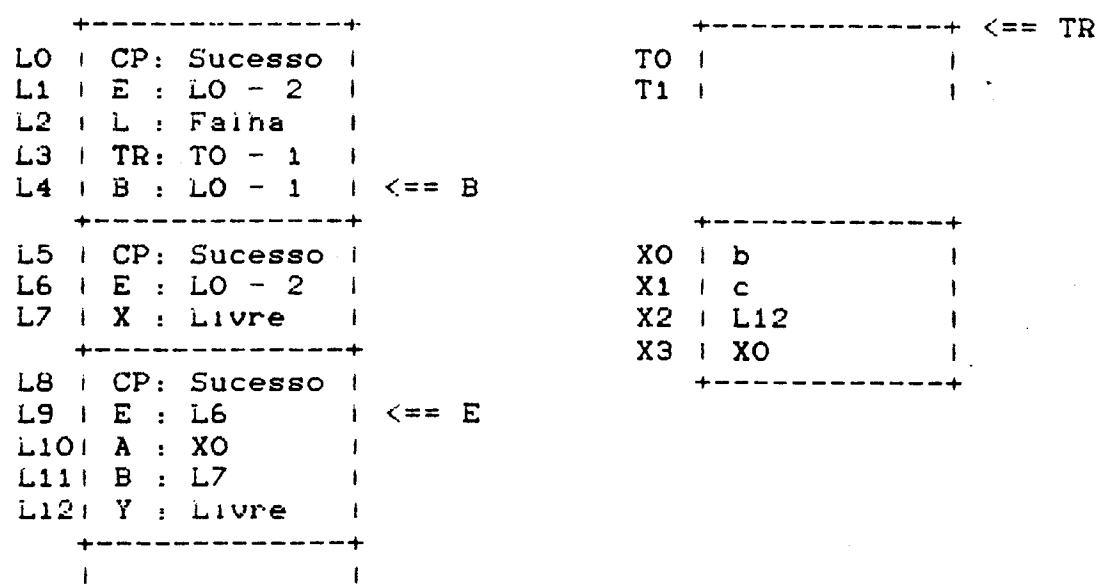
Exemplo:

```
cons( C, R, [C|R] ).

coloca( A, B ) :-
        cons( Y, A, B ),
        pega( Y ).

pega( a ).
pega( b ).

?- coloca( [ b, c ], X ).
```

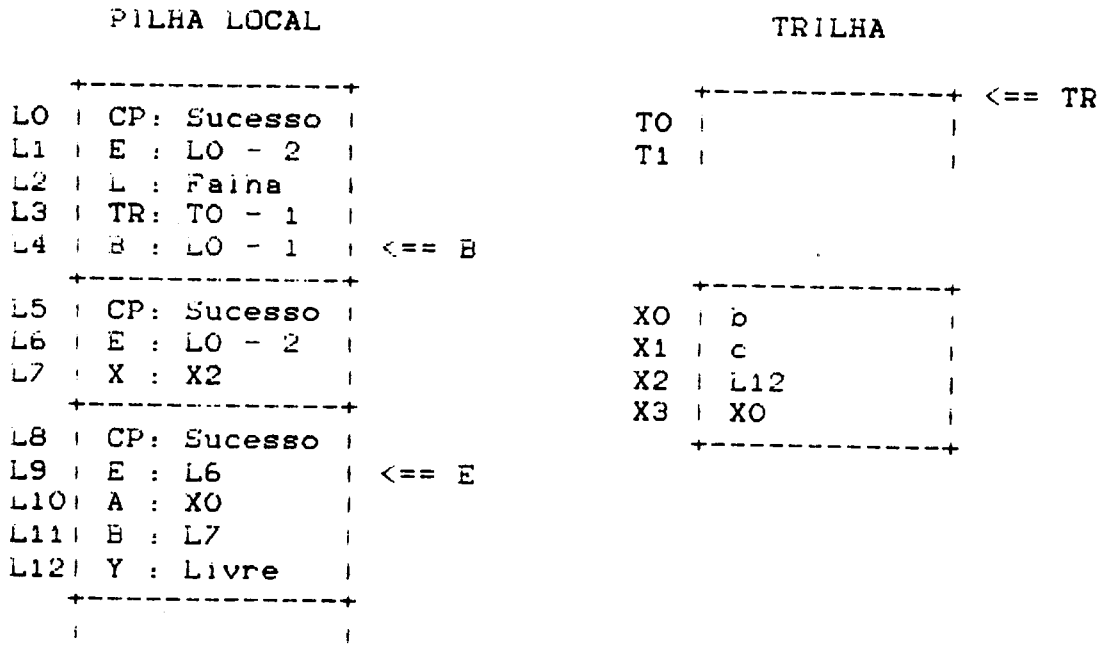
(1) Configuração inicial na ativação de "coloca":



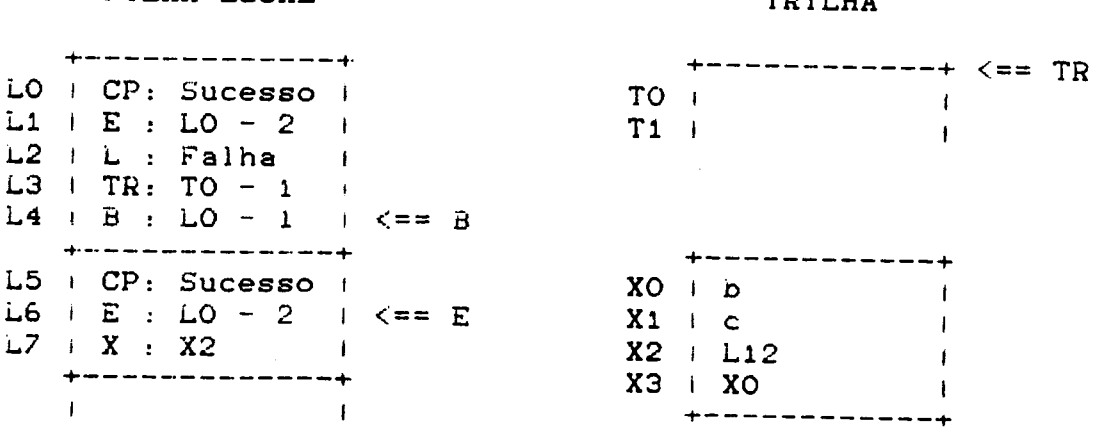
(2) Após entrarmos em "coloca":



(3) Após unificarmos "cons", criamos uma nova lista e fazemos "X" de "coloca", alocado em L7, apontar para a lista. A tática apresentada chama-se cópia de estrutura visto que cada ativação de "cons" criaria uma nova lista totalmente independente da lista anterior. Outra tática também utilizada, mas que não adotaremos, chama-se compartilhamento de estruturas, em que as partes constantes não precisam ser copiadas, diminuindo a demanda de memória mas piorando a eficiência no acesso aos componentes da estrutura.



(4) Ao desalocarmos o ambiente de "coloca" e ativarmos o procedimento "pega", temos que a variável "X" da pergunta está instanciada com uma lista (X2) na qual um dos elementos (a cabeça) é uma referência para uma variável (L12) do ambiente de "coloca", que não existe mais. Criamos assim uma referência pendente.



Existem três casos que podem criar referências pendentes: a unificação de duas variáveis livres, a unificação de uma variável livre com uma estrutura e a unificação da variável livre com algum argumento livre da estrutura.

No primeiro caso, ao unificarmos duas variáveis livres, colocamos na mais recente uma referência para a mais antiga, garantindo assim que ao desalocarmos molduras não deixaremos ponteiros inválidos.

Ao unificarmos uma variável com uma estrutura, não podemos colocar na estrutura uma referência para a variável, fazemos então o contrário, colocamos na variável uma referência para estrutura. Logo, não podemos garantir, pelo mecanismo acima descrito, que não criaremos referências pendentes. A solução encontrada é a criação de uma área de alocação dinâmica (Heap) que conterá as estruturas, eliminando assim o problema de unificação das variáveis com estruturas.

Para eliminarmos o problema de unificação entre campos de estrutura e variáveis livres basta que modifiquemos o procedimento descrito para unificação de duas variáveis livres, de forma a considerar toda área de alocação dinâmica como mais antiga que a Pilha Local. Assim, a variável unificada conterá um ponteiro para o campo da estrutura.

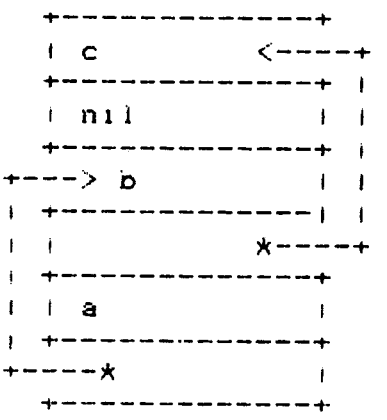
3.4.1. Organização e Funcionamento da Pilha Global.

A alocação de área dinâmica para as estruturas ocorre no processo de unificação. Quando ocorrer um retrocesso, tudo que foi criado pela unificação após aquele ponto de escolha pode deixar de existir, inclusive as estruturas. Logo, devemos saber que áreas foram alocadas antes e após a criação do ponto de escolha. Da mesma forma, quando estivermos unificando duas variáveis na área dinâmica, necessitamos saber qual a mais antiga, evitando referências pendentes.

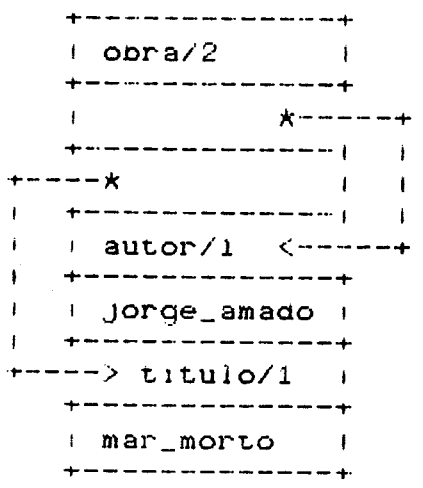
Pelas condições que acabamos de apresentar, devemos organizar a área dinâmica na forma de uma pilha, a que chamaremos de Pilha Global. Desta forma, criamos também um novo registro, H, que conterá a próxima posição livre na Pilha Global. Uma cópia de H deverá constar dos pontos de escolha de modo que em caso de retrocesso possamos liberar áreas entre o topo atual, H, e o topo no momento de criação do ponto de escolha. Analogamente à Pilha Local, variáveis livres da Pilha Global, mais ao topo que o registro H salvo no último ponto de escolha, não precisam ter suas referências colocadas na Trilha, uma vez que em caso de retrocesso estas deixarão de existir. Para implementar essa otimização necessitamos consultar o registro H salvo no último ponto de escolha. Em vez disso, criamos um novo registro, HB, que conterá esta informação.

Em um esquema de compartilhamento as estruturas seriam representadas por um ponteiro para sua parte fixa, esqueleto, e uma sequência de células que preencheriam as partes variáveis da estrutura. Duas estruturas criadas pela mesma cláusula compartilhariam o mesmo esqueleto, possuindo cada uma, seu conjunto de células. No nosso caso, copiamos a estrutura integralmente, tanto a parte fixa como a variável, a cada nova ativação da cláusula. Esse gasto adicional de memória é compensado por uma maior simplicidade no acesso. As estruturas se organizam na Pilha Global em posições contíguas, sendo a primeira um ponteiro para o funtional/aridade e as demais os seus argumentos. No caso de listas não existe o ponteiro para o funcional.

Exemplo: [a | [b | [c | []]]].



Exemplo: obra(autor(jorge_amado), titulo(mar_morto)).



3.5. Corte.

A função do corte é eliminar todos os caminhos alternativos desde a ativação do predicado que o contém até o ponto em que este aparece. Isto se traduz em eliminarmos pontos de escolha de modo que o retrocesso não os considere. Assim sendo, será necessário guardarmos em um registro, CR, o último ponto de escolha criado antes da entrada em um procedimento. Este registro deve ser salvo nos pontos de escolha e nos ambientes.

4. Conclusão.

Como visto nessa primeira parte, Prolog apresenta alguns pontos semelhantes e outros bastante diferentes das linguagens utilizadas convencionalmente, tanto no seu aspecto externo, como no seu funcionamento. Um programa em Prolog apresenta-se como uma descrição da lógica do algoritmo em questão, ficando praticamente toda a parte de fluxo de execução sob responsabilidade da linguagem. Devido a isso, a "máquina" Prolog demanda um esforço muito maior no sentido de propiciar os mecanismos aqui apresentados.

5. Referências.

[ARIT84] Arity/Prolog :- The Programming Language,
Arity Corporation, 1984.

[CLOC81] Programming in Prolog,
William F. Clocksin e Chirstopher S. Mellish,
Springer-Verlag Berlin Heidelberg, 1981.

[TICK84] Evan Tick,
Sequential Prolog Machine: Image and Host Architetures,
ACM Sigmicro Newsletter,
New York, IEEE, 15(4): 204-216, dezembro de 1987,
The Seventeenth Annual Microprogramming Workshop,
New Orleans, 1984.

[WARR83] David H. D. Warren,
An Abstract Instruction Set,
Technical Note 309,
SRI International,
Menlo Park, California, outubro de 1983.